Simone Goggian

Lecture Summary (2014)

Note Due to the nature of Eiffel some aspects mentioned here only apply to Eiffel.

Note Topic which are (more extensively) covered in other lectures (mostly Discrete Mathematics and Data Structures and Algorithms) are reduced to a minimum, mostly highlighting Eiffel-specific aspects.

MISSING: GENERAL RECURSION STRUCTURE (BY NADIA)

Contents

1 Overview 2
2 Objects 2
3 Conventions 3
4 Interfaces 3
5 Logic 3
6 Creation of Objects 4
7 References & Assignments 4
8 Control Structures 4
9 Abstraction 5
10 Dynamic Model 5
11 Inheritance 5
12 Recursion 6
13 Data Structures 6
14 Multiple Inheritance 7
15 Topological Sort 7
16 BNF 7
17 Agents 7
18 Undo/Redo 8
19 Software Engineering 8
Assignments 8
Assignment 2 8
Assignment 3 8
Exercise Sessions 8
Session 4 8
Session 5 8
Session 7 8
Further Resources 8

1 Overview

- "Ihr Computer tut genau das, was in Ihrem Programm steht."
- source code Compiler machine code
- General structure, includes everything:

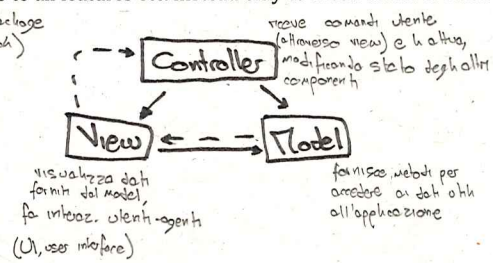
```
[deferred] class name

    [inherit name(s) [redefine name(s)]
    [undefine name(s)] end]
    [rename name as name end]
    [select name end]
    end] -- for the inherit block

    create name of creation procedure(s)

    feature [export flags]
        name [alias alias] [assign name]
            [require [else] assertions(s)]
            [local assign local variables]
            do
            feature body
            [ensure [then] assertion(s)]
            end
end
```

```
if c then
  ⋮
else
  ⋮
end
```

```
from i:=10 until i<1 loop
  ⋮
  i:=i-1
end
```

2 Objects

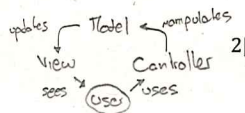
- Use Pascal case "lorem_ipsum_dolor_sit_amet" // no camelCase
- Indentation is done using tabs
- Comments use "-- text"
- Assignments use :=, reference comparisons[1] =, object comparisons ~
- Class names are written in capital case
- To call a method, use "*Objectname.featurename*"; the object is called "target" of the call
- There are three types of features (command-query separation principle; asking a question shouldn't change the answer):
  - Commands ("Befehl") (doesn't return a result), should modify one or more objects
    e.g `make (a_name: STRING; a_position: VECTOR; a_city: CITY)`
  - Queries ("Abfrage") (returns a result using **Result**), shouldn't modify nor the target or any other objects
    e.g.[2] `connecting_lines (a_station_1, a_station_2: STATION): V_SEQUENCE [LINE]`
    return type
  - Creation procedures (constructors)
- On one hand, an objects stores data and on the other hand it is a machine offering operations (commands (to modify state), queries (to access state))
- Information hiding: the client shouldn't have access to all features etc. instead only to those which are relevant
- MVC – Model, View, Controller


Controller
View
Model
riceve comandi utente (attraverso view) e li attua, modificando stato degli altri componenti
visualizza dati forniti dal model, fa interag. utente-agent (UI, user interface)
fornisce metodi per accedere ai dati utili all'applicazione

• public / private, protected / package (both / self, children / pack)

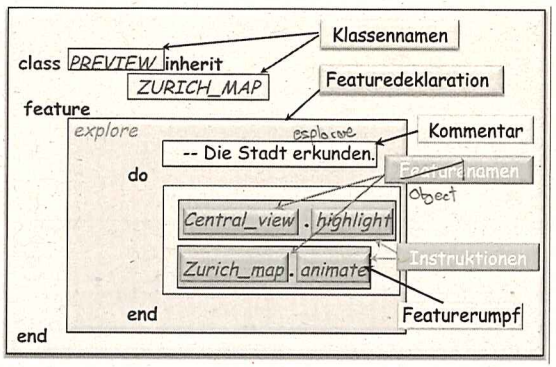
like int, double...

[1] For expanded types, this compares by value
[2] Note the generics at the end



3 Conventions

- Basic distinctions
  - Command/query: see previous lecture
  - Instruction/expression: the basic operations of a computer/program are called instructions[3]; an expression is an element of the program describing possible runtime values
  - Syntax/semantics: syntax is the form and structure of code (i.e. how you write the program), the semantics are the set of properties of its possible executions (i.e. what you expect the program to do)



- Natural language allow you to express much more while programming languages are much more precise (and extend the mathematical notation)
- Specimens ("Exemplare") are syntactic elements such as class names, instructions, one of the boxes in above figure, the whole class text, ... They can be nested but delimiter such as keyword and semicolons, periods aren't specimens
- Constructs ("Konstrukte"): each syntactic element is a specimen of a certain construct
- Abstract syntax tree (AST) showing the structure of the syntax using only specimens
- The basics elements of the source code are tokens: terminals: identifiers and constants; keyword; special symbols
- Semantic rules are based on syntactic rules which are based on lexical rules
- Identifiers start with a letter followed by one or more alphanumeric character or an underline "_"

4 Interfaces

- Definition class: A class is the description of the set of possible runtime objects to which the same features are applicable
- An object O generated by class C is called an instance of C and C is the generating class of O.
- Classes only exists in the source code, objects only exist at runtime
- An object has an interface, provided by its generating class, and an implementation, defined by its generating class.
- Contracts
  - Precondition: **require** *tag: condition*, tag & condition together are called assertion; the client has to ensure the precondition is satisfied before the call; preconditions are requirements for the client
  - Postcondition: **ensure** *tag: condition*, postconditions are benefits for the client; the feature has to ensure the postcondition is satisfied (as long as the preconditions was met, too); you can use **old** in postconditions (but nowhere else) to get the value of an expression it had when the routine was called
  - Class invariant: express consistency conditions which have to be satisfied between queries to the class

5 Logic

- Boolean values: true, false
- (semi-strict*; define an order of evaluating the expressions) Boolean operators: not (¬), or (∨), and (∧), = (⇔), implies*, /= (≠), and then*, or else*...
- Truth table for n variables is of size $(n+1) \times 2^n$
- Tautology: an expression which is always true, no matter what the values are
- Contradiction: an expression which is always false, no matter what the values are
- $a \rightarrow b$ (implies) is only false of a is true, b is false; $a \rightarrow b \Leftrightarrow \neg b \rightarrow \neg a$

[3] And, in Eiffel, can be separated by a semicolon, but don't have to be

6 Creation of Objects

- **create** *name[:creation_procedure]*[4]
- an identifier describing a runtime value is called an entity
- an entity whose value may change is a variable
- during the execution, entities can be attached to objects
- at runtime a reference is either attached to an object or **void**
- before an object is created, the precondition of the creation procedure has to be satisfied, after the creation procedure, the object isn't void anymore, the postconditions of the creation procedure is satisfied and class's invariant is satisfied.
- There is a root object/class/creation procedure which initializes the whole process
- A system/program should easily be extendible and reusable

7 References & Assignments

- An objects consists of fields, each field has either a value (expanded values[5] are a reference on another object)
- Reference type: the entity's value is a reference
- Expanded type: the entity's value is an object (**expanded class**)
- Default values of objects, variables, **Result**: 0, null, false, void
- Assign using :=
- Qualified call: x.f(a), unqualified call: f(a); unqualified calls use the current object, a qualified call uses the explicitly attached x as the current object for the call (and reverts back after the call); to use the current object explicitly, use **Current**
- When assigning a reference type value, the assignment copies the reference; when assigning an expanded type value, the assignment copies the object
- Garbage collection: soundness (only unreachable objects are GC'ed), completeness (all unreachable objects are GC'ed)
- You can (and often do) link objects together, e.g. linked lists

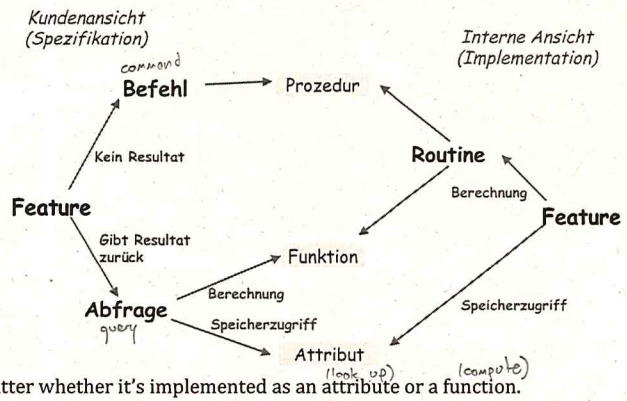
8 Control Structures

- Properties of an algorithm: 1) defines data, to which the process is applied to, 2) each elementary step is selected from a set of well-defined actions, 3)the order of execution is described, 4) (2) & (3) are based on well-defined, machine-friendly conventions, 5) terminates for every data after a finite amount of steps
- Fundamental control structures: sequence/compound[6], loop, conditional
- If/else/elseif: **if** *condition* **then** *instruction* [**elseif** *condition* **then** *instruction* [...]] [**else** *instruction*] **end**
- switch/case: **inspect** *input* **when** *value* **then** *instruction* [...] **else** *instruction* **end**
- for-loop: **from** *initialization* [**invariant** *expression*] **until** *exit condition*[7] **loop** *body* [**variant** *expression*] **end**
- across-loop: **across** *structure* **as** *variable* **loop** *instructions* **end**
- loop invariant hold before and after the exetuon of the loop body
- loop variant: integer expression that is nonnegative after execution of from clause and after each execution of the loop clause and strictly decreases with each iteration
- to iterate over a list with an internal pointer: from list.start until list.after loop list.forth end
- to iterate over a list with an external pointer: from c := list.new_cursor until c.after loop c.forth end OR across list as c loop ... end
- Halting-Problem: "Given a description of an arbitrary computer program, decide whether the program finishes running or continues to run forever"[8]
- There's also another control structure, "goto", but it's generally discouraged

[4] Often named "make", default (inherited) is default_create
[5] Int, char, real, boolean ... NOT string (!)
[6] Non-OOP is also called sequential programming
[7] *Exit* condition; loop stops when the condition is true, not vice versa (!)
[8] Wikipedia

9 Abstraction

- Data abstraction: class; abstraction of an algorithm: routine/method(/subprogram/subroutine)
- A feature can either be a routine or an attribute
- A routine is either a procedure not returning a result (command with instructions) or a function yield a result (query with expressions)
- Uniform access principle: the client doesn't care whether you have to calculate something or look it up in memory i.e. The client should be able to call a query in the same way no matter whether it's implemented as an attribute or a function.



- To implement information hiding, you can use setters (and getters[9]) for write (and read) access
- When exporting an attribute in Eiffel, you only export read access and you can't tell whether it's an attribute or a function.
- If you still want to be able to use assignments instead of setters you can declare an assigner; "temperature: REAL **assign** set_temperature" results in "x.temperature := 21.5" to be equal to "x.set_temperature(21.5)" (including its contracts etc.)
- Information hiding: every feature block (and creation procedure) can declare to which classes its features are visible to; say you are in class A: "" (nothing): available everywhere, {NONE}: invalid everywhere (also in A), {B,C}: valid in B,C and their descendants, {A,B,C}: available in A,B,C and their descendants, {A}: available in A and its descendants
- Information hiding only applies to qualified calls; unqualified calls aren't affected (otherwise {NONE} wouldn't make much sense)

10 Dynamic Model

- Variants of assignments and copying: reference assignment, flat copy, flat twin, deep twin, deep copy
- Every class implicitly inherits from ANY
- Test an object's type: **attached**{*TYPE*} *expression* **as** *"alias"*[10]

11 Inheritance

- Notation: *: deferred, +: effective, ++:redefinition
- If you picture a class in a 2D-space, you can move it along one axis to change its genericity and along the other axis you can change its abstraction/specialization i.e. inheritance "level"
- Genericity: unrestricted: LIST[**G**], restricted: HASH_TABLE[G → **HASHABLE**]; e.g. cities: LIST[**CITY**][11]; very useful for containers e.g. arrays, lists, trees, ...
- To ensure type safety you can (incomplete) either cast its type or parameterize the class (like in Eiffel)
- Static typing: type-safe call, type checker to ensure type-safety, statically typed language
- A type is either a class C whereas C is non-generic, or in D[T] where D is the name of a generic class and T is the type.
- Classes are of a dual nature: a class Is a module (grouping of related services, enforces information hiding, and has clients and suppliers) and a type (or a template, genericity) (instances of a type/runtime values, and declares entities) at the same time.
- Eiffel has multiple inheritance. **inherit**

[9] Not really needed in Eiffel, just mentioned for completeness
[10] For the lack of a better word
[11] Generic derivative

- Say B inherits from A; all services from A are also available in B; whenever the program expects an instance of A, an instance of B is valid, too ("is-a relationship").
- Terminology: Erbe, Vorgänger, Nachkommen, echte Nachkommen, Vorfahren, echte Vorfahren, direkte Instanzen, Instanzen
- Assignment: *target* := *expression*: „expression" can be either a type of "target" or its descendants; polymorphism
- An assignment (or passing of an argument) is polymorph if source and target have different (yet compatible) types
- Static type: the type of an entity in the source code; dynamic type: the type of the object the entity is attached to (at runtime); The dynamic type always conforms to the static type. (For a polymorphic attachment to be valid, the type of the source must conform to the type of the target.)
- If a class D is a descendant of a class C, both non-expanded, then types derived from D conform to those derived from C as follows: If the classes are not generic, then D(as a type) conforms to C; If they are generic, then D [T,U, ...] conforms to C [T,U, ...](with the same generic parameters).
- An expanded type conforms only to itself.
- Redefinition: a class can redefine an inherited feature; **redefine**
- Dynamic binding: every execution of a feature call calls the feature which is best adapted to the target object.
- Static typing guarantees at least one version of feature f exists, dynamic typing guarantees every call calls the most well-suited version of feature f is called.
- In an inheritance hierarchy, every class is either effective, deferred, or redefined.
- Deferred class: can have deferred features (even if there's just one deferred feature, the while class has to be declared deferred), which don't have a body (and thus cannot be instantiated)
- Contracts: when redeclaring a routine, the precondition has to be either kept or weakened while the postcondition has to be either kept or strengthened. **require else**, **ensure then**, this reulsts in the following: original_precondition *or* new_precondition, orirignal_postcindtion *and* new_postcondition

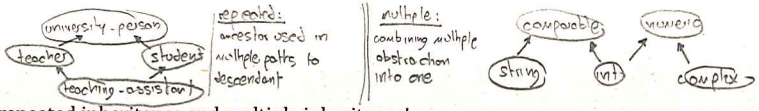
12 Recursion

- Direct and indirect recursion: direct: r calls s, s calls r; indirect: a calls b, b calls c, c calls d, d calls a
- Recursive routine: the routine's body contains a call to the routine itself
- Recursion can be used to traverse a binary tree
- A useful recursive definition ensures the following: 1) there's at least one non-recursive branch, 2) each recursive branch appears in a different context than the original context, 3) for every recursive branch, the context change in (2) should be bring it closer to a non-recursive branch (1)
- Recursion is often a loss of performance
- Example: divide-and-conquer

13 Data Structures

- A container provides the following operations: insert, delete, wipe, search, traverse/iterate
- Standardized names in EiffelBase: is_empty, has, count, item, make, put, remove, wipe_out, start, finish, forth, back
- Examples: list, linked list, doubly linked list; array, arrayed list; hash tables
- For arrays, the following pairs are equivalent: a[i], a.item(i); a.item(i) := x, a.put(x,i); a[i]:=x, a.put(x,i)
- Always keep the edge cases in mind (full, empty)
- Open/closed/perfect hash5ng
- Dispensers: LIFO (stack), FIFO (queue), priority queue
- Use cases
  - Linked list: order matters, most accesses are in that order, no fixed size limit
  - Array: each element can be indexed with an integer, most accesses use this index, fixed size limit
  - Hashtable: each item has a key, most accesses use that key, the structure is limited
  - Stack: LIFO, e.g. traversing a tree
  - Queue: FIFO, e.g. simulating FIFO

14 Multiple Inheritance



- Mind the difference between repeated inheritance and multiple inheritance!
- Resolve name conflicts: **rename** (keep the feature under a different name), **redefine** (keep the name but change the feature)
- Name conflicts must be resolved unless the conflict is due to repeated inheritance (not a real conflict) or when only one feature is effective (and the others are deferred).
- Acceptable name conflicts: if inherited features all have the same name, there's no problem if they have a compatible signature and at most one is effective.
- To resolve ambiguity, one feature can be **selected**

15 Topological Sort

- To create a total order out of a given partial order.
- Examples: glossary where every word is defined before it's used; schedule/order of execution
- Doesn't work with cyclic dependencies (ensure no cycles)
- A relation can have the following properties: reflexive, irreflexive, symmetric, anti-symmetric, asymmetric, transitive

16 BNF (Backus normal form)

• descrive sintassi linguaggio in maniera specifica, non ambigua.

- compilers need a strictly formal definition of a programming language
- a language is a set of phrases; a phrase is a finite sequence of tokens of a certain vocabulary; Not every possible sequence is a phrase; A grammar specifies which sequences are part of the language and which not; BNF is used to define a grammar for a programming language
- A grammar is finite set of rules to create sequences of tokens which ensures, that every sequence, created by applying grammar rules finitely often, is a phrase of the language, and every phrase of the language can be created by applying the rules of the grammar finitely often.
- Terminals: characters which aren't defined by the grammar, e.g. keywords and symbols in Eiffel.
- Non-terminals: names of syntactic (sub-) structures used to create phrases.
- Productions: rules which using a combination of terminals and non-terminals define the non-terminal of a grammar.
- Examples: compound $A\ B$; optional $[A]$, selection: $A|B$, repetition (0 or more) $\{A\}^*$, repetition (at least once) $\{A\}^+$

17 Agents

- Aka delegates, closures
- Good application: GUIs
- From the .NET docs: Events have the following properties:
  - 1. The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
  - 2. An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
  - 3. Events that have no subscribers are never called.
  - 4. Events are commonly used to signal user actions such as button clicks or menu selections in graphical user interfaces. (GUI)
  - 5. When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see [another section].
  - 6. Events can be used to synchronize threads.
  - 7. In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class.
- Publisher & subscriber / publish & subscribe
- Event-oriented/driven design & programming

- Event-context-action table approach: a set of triples defining the event type, the context and the action; one generic class EVENT_TYPE with features publish and subscribe; use of *subscribe*: "subscribe(**agent** *agent_name*)"
- A function called as an agent (in the subscriber method), can use a question mark "?" as a placeholder which is then an open argument (opposed to a close argument)
- To actually call the function, you use a.call([*arguments*])[12]; if a is associated with a function, a.item([*arguments*]) returns the result
- Declaring an agent is one of the following three: PROCEDURE[ANY, TUPLE] is a procedure with no open arguments; PROCEDURE[ANY, TUPLE[X,Y,Z]] is a procedure with three open arguments; FUNCTION[ANY, TUPLE[X,Y,Z], RES] is a procedure with three open arguments and a result of type RES
- Agents can also be inline, using an assignment to a variable; **agent signature do** *feature body* **end**

18 Undo/Redo[13]

19 Software Engineering[14]

Assignments

Assignment 2

- To write to the console, use Io.put_string and Io.new_line; to read use Io.read_string

Assignment 3

- Difference object/class: A class can be viewed as a software module (a piece of source code that contains descriptions of related operations and data), and a type (a set of objects that support the same operations). An object can be viewed as a collection of data (whose structure is defined in the object's generating class) and a member of a type (an entity in a running program, to which the operations defined in the generating class are applicable).
- Real world analogy class/object: A class can be viewed as the blueprint of a machine, while an object is the actual machine built according to the blueprint.

Exercise Sessions

Session 4

- Expanded classes are not creatable using a creation feature but have to use (and redinfe) default_create
- a,b:VECTOR; create b.make(1.0,1.0);a:=b;; now a and b reference the same object (aliasing) and thus when changing a you also "change" b (and vice versa).

Session 5

- Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way. (otherwise use feature)
- Attributes can be constants
- Entity can be any of the following: attribute name, variable attribute, constant attribute, formal argument name, local variable name, **Result**, **Current**

Session 7

- If a feature was redefined, but you still wish to call the old one, use the **Precursor** keyword.

Further Resources

- http://wiki.vis.ethz.ch/Zusammenfassung_Einf%C3%BChrung_in_die_Programmierung

[12] Note the square brackets: they aren't meant to denote optional arguments but are part of the syntax (!!)
[13] omitted
[14] omitted